FIPS PUB 106

FEDERAL INFORMATION
PROCESSING STANDARDS PUBLICATION

1984 JUNE 15

U.S. DEPARTMENT OF COMMERCE/National Bureau of Standards

GUIDELINE ON SOFTWARE MAINTENANCE

EGORY: SOFTWARE
CATEGORY: SOFTWARE MAINTENANCE

FIPS PUB 106

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
National Bureau of Standards, Ernest Ambler, *Director*

Foreword

The Federal Information Processing Standards Publication Series of the National Bureau of Standards is the official publication relating to standards adopted and promulgated under the provisions of Public Law 89-306 (Brooks Act) and under Part 6 of Title 15, Code of Federal Regulations. These legislative and executive mandates have given the Secretary of Commerce important responsibilities for improving the utilization and management of computers and automatic data processing in the Federal Government. To carry out the Secretary's responsibilities, the NBS, through its Institute for Computer Sciences and Technology, provides leadership, technical guidance, and coordination of government efforts in the development of guidelines and standards in these areas.

Comments concerning Federal Information Processing Standards Publications are welcomed and should be addressed to the Director, Institute for Computer Sciences and Technology, National Bureau of Standards, Gaithersburg, MD 20899.

James H. Burrows, *Director*
Institute for Computer Sciences and Technology

Abstract

There is a need for a strong, disciplined, clearly-defined approach to software maintenance. This report emphasizes the importance of the consideration of software maintenance throughout the lifecycle of a software system and stresses the need to plan, develop, use, and maintain a software system with future software maintenance in mind. General and functional definitions of software maintenance are provided and software change activities are identified. The report presents guidance for controlling and improving the software maintenance process and includes suggested criteria for deciding whether continued maintenance of a software system is justified. It concludes that an organization's software maintenance efforts can be improved through the institution and enforcement of software maintenance policies, standards, procedures, and techniques.

Key words: adaptive maintenance; corrective maintenance; Federal Information Processing Standards Publications; management; perfective maintenance; software engineering; software lifecycle; software maintenance; software maintenance management; software maintenance tools.

Natl. Bur. Stand. (U.S.) Fed. Info. Process. Stand. Publ. (FIPS PUB) 106, 20 pages
(1984)
CODEN:FIPPAT

For sale by the National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Federal Information Processing Standards Publication 106

1984 June 15

ANNOUNCING THE

GUIDELINE ON SOFTWARE MAINTENANCE

Federal Information Processing Standards Publications are issued by the National Bureau of Standards pursuant to the Federal Property and Administrative Services Act of 1949, as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973), and Part 6 of Title 15 Code of Federal Regulations (CFR).

Name of Guideline: Guideline on Software Maintenance.

Category of Guideline: Software; Software Maintenance.

Explanation: This Guideline presents information on techniques, procedures, and methodologies to employ throughout the lifecycle of a software system to improve the maintainability of that system. Guidance is presented on controlling and improving software maintenance. Also included is a glossary of technical terms, a list of supporting ICST publications, and a list of suggested additional reading. Appendices provide information on the software maintenance process; how to decide whether or not to continue maintaining a system; and software maintenance tools. This Guideline is intended for use by both managers and maintainers.

Approving Authority: U.S. Department of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology.

Maintenance Agency: U.S. Department of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology.

Cross Index: None.

Applicability: This Guideline is intended as a basic reference guide for Federal ADP managers and software maintainers for maintaining software systems throughout their lifecycle. It addresses both management and technical issues. Use of this Guideline is encouraged, but not mandatory.

Implementation: This Guideline should be consulted whenever Federal departments or agencies are: developing or maintaining software; developing policies and procedures for developing or maintaining software; or considering alternatives to continued maintenance of a software system.

Specifications: Federal Information Processing Standards Publication 106 (FIPS PUB 106), Guideline on Software Maintenance (affixed).

Qualifications: The techniques and procedures presented in this Guideline are recommended for use in all Federal ADP organizations. Specific environments, organizational priorities, available budget and staff resources, and many other factors should be taken into account when implementing these recommendations. The resulting organizational standards and/or guidelines should reflect the specific needs of the organization.

Where To Obtain Copies: Copies of this publication are for sale by the National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. When ordering, refer to Federal Information Processing Standards Publication 106 (FIPSPUB106), and title. When microfiche is desired, this should be specified. Payment may be made by check, money order, or NTIS deposit account.

Federal Information
Processing Standards Publication 106

1984 June 15

Specifications for

GUIDELINE ON SOFTWARE MAINTENANCE

Contents

Figures

1. INTRODUCTION

Software maintenance accounts for approximately 60% to 70% of the application software resources expended within the Federal Government. In addition, the rapidly growing inventory of software systems is increasing the demand for software maintenance. Improved productivity in maintaining software, however, can offset these increases. Thus, the issue which must be addressed is not reducing the absolute cost, but rather improving the quality and effectiveness of software maintenance.

This Guideline provides general guidance for managing software maintenance. It presents an overview of techniques and procedures designed to assist management in controlling and improving the software maintenance process. It addresses the need for a software maintenance policy with enforceable controls for use throughout the software lifecycle, the management of software maintainers, and management methods which can improve software maintenance. It concludes that improvements in the area of software maintenance will come primarily as a result of the software maintenance policies, standards, procedures, and techniques instituted and enforced by management.

This Guideline is intended for use by both managers and maintainers. It addresses the need for a strong, disciplined, clearly defined approach to software maintenance. It emphasizes that the maintainability of the software must be taken into consideration throughout the lifecycle of a software system. Software must be planned, developed, used, and maintained with future software maintenance in mind. The techniques and procedures which are discussed in this Guideline are recommended for use in all Federal ADP organizations. Specific environments, organizational priorities, available resources (budget and staff), and many other factors also must be taken into account, however. The resulting organizational standards and guidelines should reflect the specific needs of the organization.

This Guideline is divided into five sections and two appendices:

Section 1—INTRODUCTION discusses the purpose of this Guideline.

Section 2—SOFTWARE MAINTENANCE DEFINITION presents general and functional definitions of software maintenance.

Section 3—THE SOFTWARE MAINTENANCE PROCESS provides a correlation between the software lifecycle and a software maintenance lifecycle, and identifies software change activities.

Section 4—CONTROLLING AND IMPROVING SOFTWARE MAINTENANCE provides the actual guidance on techniques and procedures to aid both management and the maintainer.

Section 5—SOFTWARE MAINTENANCE VS SYSTEM REDESIGN discusses factors which should be considered when deciding whether to re-develop or to continue maintenance of a software system.

Appendix I lists related reports published by the Institute for Computer Sciences and Technology of the National Bureau of Standards. Appendix II provides references to selected readings on software maintenance. A Glossary of frequently used software maintenance terms is also provided.

2. SOFTWARE MAINTENANCE DEFINITION

Software maintenance is the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production.

Software maintenance is the set of activities which result in changes to the originally accepted (baseline) product set. These changes consist of corrections, insertions, deletions, extensions, and enhancements to the baseline system. Generally, these changes are made in order to keep the system functioning in an evolving, expanding user and operational environment.

Functionally, software maintenance activities can be divided into three categories: perfective, adaptive, and corrective.

Perfective maintenance includes all changes, insertions, deletions, modifications, extensions, and enhancements which are made to a system to meet the evolving and/or expanding needs of the user. Perfective maintenance refers to enhancements made to improve software performance, maintainability, or understandability. It is generally performed as a result of new or changing requirements, or in an attempt to augment or fine tune the software. Activities designed to make the code easier to understand and to work with, such as restructuring or documentation updates (often referred to as "preventive" maintenance) and optimization of code to make it run faster or use storage more efficiently are also included in the perfective category. Perfective maintenance comprises approximately 60% of all software maintenance.

Adaptive maintenance consists of any effort which is initiated as a result of changes in the environment in which a software system must operate. These environmental changes are normally beyond the control of the software maintainer and consist primarily of changes to the:

- rules, laws, and regulations that affect the system
- hardware configurations, e.g., new terminals, local printers
- data formats, file structures
- system software, e.g., operating systems, compilers, utilities.

Approximately 20% of software maintenance falls into the adaptive category.

Corrective maintenance refers to changes necessitated by actual errors (induced or residual "bugs") in a system. Corrective maintenance consists of activities normally considered to be error correction required to keep the system operational. By its nature, corrective maintenance is usually a reactive process where an error must be fixed immediately. Not all corrective maintenance is performed in this immediate response mode; but all corrective maintenance is related to the system not performing as originally intended. Corrective maintenance accounts for approximately 20% of all software maintenance. The three main causes of corrective maintenance are: (1) design errors, (2) logic errors, and (3) coding errors.

3. THE SOFTWARE MAINTENANCE PROCESS

The lifecycle of computer software covers its existence from its conception until the time it is no longer available for use. There are a number of definitions of the software lifecycle which differ primarily in the categorization of activities or phases. One traditional definition is: *requirements, design, implementation, testing,* and *operation and maintenance.*

The *requirements phase* encompasses problem definition and analysis, statement of project objectives, preliminary system analysis, functional specification, and design constraints. The *design phase* includes the generation of software component definition, data definition, and interfaces which are then verified against the requirements. The *implementation phase* entails program code generation, unit tests, and documentation. During the *test phase,* system integration of software components and system acceptance tests are performed against the requirements. The *operations and maintenance phase* covers the use and maintenance of the system. The beginning of the maintenance phase of the lifecycle is usually at the delivery and user acceptance of the software product set.

One way of describing the activities of software maintenance is to identify them as successive iterations of the first four phases of the software lifecycle, i.e., *requirements, design, implementation,* and *testing*. Software maintenance involves many of the same activities associated with software development but also has unique characteristics of its own.

1. Maintenance activities are performed within the context of an existing framework or system. The maintainer must make changes within the existing design and code structure constraints. The older the system, the more challenging and time-consuming the software maintenance effort often becomes.

2. A software maintenance effort is typically performed within a much shorter time frame than a development effort. A software development effort may span one, two, or more years while corrective maintenance may be required within hours and perfective maintenance in cycles of 1 to 6 months.

3. Development efforts must create all of the test data from scratch. Maintenance efforts typically can take advantage of existing test data and perform regression tests. The major challenge for the maintainer is to create new data to adequately test the changes to the system and their impact on the rest of the system.

The process of implementing a change to a production system is complex and involves many people in addition to the maintainer. This process begins when the need for a change arises and ends after the user has accepted the modified system and all documentation has been satisfactorily updated.

Although the software maintenance process is presented in a linear fashion in figure 1, there are a number of steps where iterative loops often occur. The change request may be returned to the user for additional clarification; the results of the design review may necessitate additional design analysis or even modification of the change request; testing may result in additional design changes or recoding; the standards audit may require changes to the design documents, code, and/or documentation; and the failure of the users to accept the system may result in return to a previous step or the cancellation of the task.

Not all of the steps presented here, however, must be performed for each change. There are several points at which the process may end. The key is to ensure that each person or group impacted by a change is involved in the process, aware of the actions taken, and satisfied with the results.

1. Determination of need for change
2. Submission of change request
3. Requirements analysis
4. Approval/rejection of change request
5. Scheduling of task
6. Design analysis
7. Design review
8. Code changes and debugging
9. Review of proposed code changes
10. Testing
11. Update documentation
12. Standards audit
13. User acceptance
14. Post-installation review of changes and their impact on the system
15. Completion of task

Figure 1. *The software maintenance process*

4. CONTROLLING AND IMPROVING SOFTWARE MAINTENANCE

Software maintenance must be performed in a structured, controlled manner. It is simply not enough to get a system "up and running" after it breaks. Proper management control must be exercised over the entire process. In addition to controlling the budget, schedule, and staff, it is essential that the software maintenance manager control the system and the changes to it.

A great deal of code is not developed with maintenance in mind. Indeed, the emphasis has often been to get the program up and running without being "hindered" by guidelines, methodologies, or other controls. In addition, over the lifecycle of a software system, code and logic which may have been well-designed and implemented often deteriorate due to an endless succession of "quick fixes" and patches which are neither well-designed nor well-documented. A system must not only be developed "with maintenance in mind," it must also be "maintained with future maintenance in mind." If this is done, the quality and maintainability of the code actually can improve. Otherwise, today's maintainable systems are destined to become tomorrow's unmaintainable systems.

4.1 Controlling Software Maintenance

The quality and maintainability of a software system often decrease as a system grows older. This is the result of many factors which, taken individually, may not seem significant. They are, however, cumulative and often result in a system which is very difficult to maintain. Quality programming capabilities and techniques are readily available. Until a firm discipline is defined for the performance of software maintenance, and that discipline is enforced, many systems will be permitted to deteriorate to the point where they are impossible to maintain.

The goal of software maintenance management is to keep systems functioning and to respond to user requests in a timely and satisfactory manner. Given the realities of staffing limitations, computer resource limitations, and the user request backlog, this goal is very difficult to achieve. The realistic goal, then, is to keep the software maintenance process orderly and under control. The specific responsibility of the software maintenance manager is to keep application systems running and to facilitate communication between management, users, and maintainers.

Controlling software maintenance is primarily maintaining an orderly process in which all requests are formally submitted, reviewed, assigned a priority, and scheduled. This does not mean that unnecessary delays should be built into the process, or that in small organizations these steps are not consolidated. Rather, it defines a philosophical approach which can help the software maintenance manager bring order to the software maintenance environment.

It is very rare for even a "perfect" system not to require significant maintenance during its lifetime. While software does not "break" in the sense that a piece of hardware can fail, it can become non-functional, or faulty due to changes in the environment in which it must operate, the size or sophistication of the user community, the amount of data it must process, or damage to code which is the result of other maintenance efforts on other parts of the system. Difficulties encountered during software maintenance can be reduced significantly by the adoption and enforcement of appropriate standards and procedures during the development and maintenance of the software.

Establish a Software Maintenance Policy

The establishment of a software maintenance policy for an organization is a vital step in controlling software maintenance. A software maintenance policy should describe in broad terms the responsibilities, authorities, functions, and operations of the software maintenance organization. It should be sufficiently comprehensive to address any type of change to the software system and its environment, including changes to the hardware, software and firmware. To be effective, the policy should be consistently applied and must be supported and promulgated by upper management to the extent that it establishes an organizational commitment to software maintenance. A software maintenance policy should direct attention toward the need for greater discipline in software design, development, and maintenance.

The software maintenance policy must specifically address the need and justification for changes, the responsibility for making the changes, the change controls and procedures, and use of modern programming practices, techniques and tools. It should describe management's role and duties in regard to software maintenance and define the process and procedures for controlling changes to the software. Implementation of the policy has the effect of enforcing adherence to rules regarding the operating software and documentation from initiation through completion of the requested change. Once this is accomplished, it is possible to establish the milestones necessary to measure software maintenance progress. Reviews and audits are required to ensure the plans are carried out. The key to controlling changes to a system is the centralization of change approval and the formal requesting of changes.

Everything done to software affects its quality. Thus, measures should be established to aid in determining which category of changes are likely to degrade software quality. Care must also be taken to ensure that changes are not incompatible with the original system design and intent. The degree to which a change is needed and its anticipated use should be a major consideration. Consideration should also be given the cost/benefit of the change: "Would a new system be less expensive and provide better capabilities?". The primary purpose of change control is to assure the continued smooth functioning of the application system and the orderly evolution of that system. Therefore, the policies establishing change control should be clear, concise, well publicized, and strictly enforced.

1. - Review and evaluate all requests for changes.
   - Require formal (written) requests for all changes.
   - Review all change requests.
   - Analyze and evaluate the type and frequency of change requests.
   - Consider the degree to which a change is needed and its anticipated use. All changes should be fully justified.
   - Evaluate changes to ensure that they are not incompatible with the original system design and intent. No change should be implemented without careful consideration of its ramifications.
   - Emphasize the need to determine whether a proposed change will enhance or degrade the system.
   - Approve changes only if the benefits outweigh the costs.
2. Plan for and schedule maintenance.
   - Assign a priority to each change request.
   - Schedule each approved change request.
   - Adhere to the schedule.
   - Plan for preventive maintenance.
3. Restrict code changes to the approved work.
4. Enforce documentation and coding standards through reviews and audits.

Figure 2. *Controlling software maintenance*

Review and evaluate all requests for changes

All user and staff requests for changes to an application system (whether enhancements, preventive maintenance, or errors) should be requested in writing and submitted to the software maintenance manager. Each change request should include not only the description of the requested change, but a full justification of why that change should be made. These change requests should be carefully reviewed and evaluated before any actual work is performed on the system. The evaluation should take into consideration, among other things, the staff resources available versus the estimated workload of the request; the estimated additional computing resources which will be required for the design, test, debug and operation of the modified system; and the time and cost of updating the documentation. Flexibility should be built into the process with some delegation of authority to initiate critical tasks when necessary. However, each request should be reviewed and evaluated by either the software maintenance manager or a change review board.

Plan for, and schedule maintenance

The result of the review of all change requests should be the assignment of a priority to each request and the updating of a schedule for meeting those requests. In many ADP organizations, there are simply more work requests than staff resources to meet those requests. Therefore, all work should be scheduled and every effort made to adhere to the schedule rather than constantly changing course in response to the most visible crisis.

Restrict code changes to the approved work

In many cases, especially when the code was poorly designed and/or written, there is a strong temptation to change other sections of the code as long as the program has been "opened up." The software maintenance manager must monitor the work of the software maintenance staff, and ensure that only the authorized work is performed. In order to monitor maintenance effectively, all activities must be documented. This includes everything from the change request form to the final revised source program listing.

Permitting software maintenance staff to make changes other than those authorized can cause schedules to slip and may prevent other, higher priority work from being completed on time. It is very difficult to limit the work which is done on a specific program, but it is imperative to the overall success of the software maintenance function to do so.

Enforce documentation and coding standards

Proper and complete communication of necessary information between all persons who have worked, are currently working, and who will work on the system is essential. The most important media for this communication are the documentation and the source code.

It is not enough to simply establish standards for coding and documentation. Those standards must be continually enforced via technical review and examination of all work performed by the software maintenance staff. In scheduling maintenance, sufficient time should be provided to fully update the documentation and to satisfy established standards and guidelines before a new assignment is begun.

4.2 Improving Software Maintenance

Maintainability is the ease with which software can be changed to satisfy user requirements or can be corrected when deficiencies are detected. The maintainability of a system must be taken into consideration throughout the lifecycle of that system. If the software is designed and developed initially with maintenance in mind, it can be more readily changed without impacting or degrading the effectiveness of the system. This can be accomplished if the software design is flexible, adaptable, and structured in a modular, hierarchical manner. Input from management, users, and developers during the design phase is also essential to improving maintainability. Such input enables the system designers to gain a better understanding of what is needed.

Many techniques and aids exist to assist the system developer, but there has been little emphasis on aids for the maintainer. However, since the processes which occur in the maintenance phase are similar to those of the development phase, there is considerable overlap in the applicability of the development aids in the maintenance environment.

The philosophies, procedures, and techniques presented here should be utilized throughout the lifecycle of a system in order to provide maintainable software. Software systems which were not developed using these techniques can benefit from their application during major maintenance activities. As a system is maintained, the maintainability of the system can be improved by applying this guidance to the parts of the system which are modified during the maintenance process. While the effect will not be as pronounced as when programs are "developed with maintenance in mind," future maintenance efforts can be made easier by utilizing these techniques to "maintain systems with future maintenance in mind."

4.2.1 Source Code Guidelines

Source code guidelines and standards aid maintainability by providing a structure and framework within which systems can be developed and maintained in a common, more easily understood, manner. Guidelines should reflect the needs and environment of a specific organization and should be based on the following basic principles:

Single high-order language

Wherever possible, a single high order language (HOL) should be used. High order languages resemble English and are generally, self-documenting. Since there are standards for most of the commonly used HOLs, it is easier to move a system written in an HOL from one environment to another.

Coding conventions

The first obstacle a maintainer must conquer is the code itself. A great deal of the source code written by developers and maintainers is not written with the future maintainer in mind. Thus, the readability of source code is often very poor. *Source code should be self-documenting and written in a structured format.*

Simple rules regarding the use of the language(s) and the physical formatting of the source code should be established. The following techniques can improve program readability and should be used as the basis for a code standard.

- *Keep it simple.* Complicated, fancy, exotic, tricky, confusing, or "cute" constructions should be avoided whenever a simpler method is available.
- *Indentation*, when properly utilized between sections of code, serves to block a listing into segments. Indentation and spacing are both ways to show subordination.
- *Extensively comment the code* with meaningful comments. Do not comment for comment's sake. Rather, comment in order to communicate to subsequent maintainers not only what was done and how it was done, but why it was done in this manner.
- Use *meaningful variable names* which convey both what the data item is and why it is used.
- *Similar variable names* should be avoided.
- *Parameters* should be used to pass data values between routines in a program.
- When *numerals* are used, they should be placed at the end of the variable name. Numbers used as program tags or labels should be sequential.
- *Logically related functions should be grouped* together in the same module or set of modules. To the extent possible, the logic flow should be from top to bottom of the program.
- *Avoid non-standard features* of the version of the programming language being used.

Structured, modular software

A structured program is constructed with a basic set of control structures or modules which each have one exit and one entry point. Structured programming techniques are well-defined methods which incorporate top-down design and implementation and strict use of structured programming constructs. Whether the strict definition, or a more general approach (which is intended to organize the code and reduce its complexity) is used, structured programming has proven to be useful in improving the maintainability of a system.

A program comprised of small, hierarchical units or sets of routines, where each performs a particular, unique function, is said to be modular. Modularity is not merely program segmentation. Modules should be constructed using the following basic design principles:

- Modules should perform only one principal function.
- Interaction between modules should be minimal.
- Modules should have only one entry and one exit point.

Standard data definitions

It is imperative that a standard set of data definitions be developed for a system. These data definitions, which may be collected in a data dictionary, should define the name, physical attributes, purpose, and content of each data element utilized in the system. These names should be as descriptive and meaningful as possible. If this is consistently and correctly done, the task of reading and understanding each module and ensuring correct communication between each module is greatly simplified.

Well-commented code

The purpose of comments is to convey information needed to understand the process and the reasons for implementing it in that specific manner, not how it is being done. Good commentary increases the intelligibility of source code. In addition to making programs more readable, comments serve two other vital purposes. They provide information on the purpose and history of the program, its origin (the author, creation and change dates), the name and number of subroutines, and input/output requirements and formats. They also provide operation control information, instructions, and recommendations to help the maintainer understand aspects of the code that are not clear. Comments are often the primary form of documentation.

Compiler extensions

The use of non-standard features of a compiler can have serious effects on the maintainability of a system. If a compiler is changed, or the application system must be transported to a new machine, there is a very great risk that the extensions of the previous compiler will not be compatible with the new compiler. Thus, it is best to refrain from language extensions and to stay in conformance with the basic features of the language. If it is necessary to use a compiler extension, its use should be well-documented.

4.2.2 Documentation Guidelines

The documentation of a system is essential to good maintenance and should start with the original requirements and design specifications and continue throughout the lifecycle of the system. The documentation must be planned so a maintainer can quickly find the needed information. Documentation should support the useable transfer of pertinent information and should include instructions on what information must be provided, how it should be structured, and where the information should be kept.

A number of methodologies and guidelines exist which stress differing formats and styles. While preference may differ on which methodology to use, it is important to adopt a documentation standard and to then consistently enforce adherence to it for all software projects. In establishing documentation guidelines and standards, keep in mind that the purpose is to communicate necessary, critical information, not to communicate all information.

The key to successful documentation is that not only must the necessary information be recorded, it must be easily and quickly retrievable by the maintainer. On-line documentation which has controlled access and update capabilities is the best form of documentation for the maintainer. If the documentation cannot be kept on-line, a mechanism must exist to permit access to the hard-copy documentation by the maintainer at any time.

Basically, the documentation standards should require the inclusion of all pertinent material in a documentation folder or notebook. There should be a requirement to complete and/or update documentation before new work assignments are begun. If documentation guidelines, or any other software guidelines or standards, are to be effective, they must be supported by a level of management high enough within the organization to ensure enforcement by all who use the software or are involved with software maintenance.

4.2.3 Coding And Review Techniques

The techniques listed in this section have been found to be very effective in the generation of maintainable systems. Not all techniques are generally applicable to all organizations, but it is recommended that they be considered.

Top down/bottom up approach

A top-down design approach (development or enhancements) involves starting at the macro or overview level and successfully breaking each program component or large, complex problem into smaller, less complicated segments. These segments are then decomposed into even smaller segments until the lowest level module of the original problem is defined for each branch in the logic flow tree.

The bottom-up design approach begins with the lowest level of elements. These are combined into larger components which are then combined into divisions, and finally, the divisions are combined into a program. A bottom-up approach emphasizes designing the fundamental or "atomic" level modules first and then using these modules as building blocks for the entire system.

In most cases a combination of top-down and bottom-up approaches should be utilized to develop a clear, concise, maintainable system.

Peer reviews

Peer review is a quality assurance method in which two or more programmers review and critique each other's work for accuracy and consistency with other parts of the system. This type of review is normally done by giving a section of code developed by one programmer to one or more other peer programmers who are charged with identifying what they consider to be errors and potential problems. It is important to establish and to keep clearly in the participants' minds that the process is not an evaluation of a programmer's capabilities or performance. Rather, it is an analysis and evaluation of the code. As stated in the name, such reviews are performed on a peer basis (programmer to programmer) and should never be used as a basis for employee evaluation. Indeed, project managers should not, if possible, be involved in the peer reviews.

Inspections

Inspection refers to a formal evaluation technique employed to identify discrepancies and to measure quality (error content) of an application system throughout the software lifecycle. The inspection generally begins with an overview that identifies procedural logic, paths, and interdependencies of the plan or deliverables. Next, the areas to be scrutinized are flagged and examined in detail to detect faults, violations of development and maintenance standards, and other problems. A report is then produced which identifies errors uncovered and requirements for error correction.

The responsibilities and activities of the inspection team, and a checklist which identifies the deliverables and materials to be examined are prepared in advance. This checklist is updated as uncovered errors are resolved and new errors found. Inspections are generally performed by a team composed of a team leader and other persons who have an indepth knowledge of the system design and functions, an understanding of the specific program areas to be inspected, and an ability to determine and establish test criteria. The team may, but normally does not, include the author of the plans, programs, or deliverables. A follow-up inspection is performed to ensure that each error has been corrected successfully. While inspections are more formal and generally require greater effort than walkthroughs, their benefits lie in the capacity to produce repeatable results at specified checkpoints.

Walkthroughs

Walkthroughs of a proposed solution or implementation of a maintenance task can range from informal to formal, unstructured to structured, and simple to full-scale. In its simplest form, a walkthrough can be two maintainers sitting down and discussing a task which one of them is working on. In its more complex forms, there may be a structured agenda, report forms, and a recording secretary. The goal is an open, frank dialogue which results in the refinement of good ideas and the changing or elimination of bad ones. Managers may or may not participate in walkthroughs.

4.2.4 Controlling Change

Change control is necessary to ensure that all software maintenance requests are handled accurately, completely, and in a timely manner. It helps assure adherence to the established standards and performance criteria for the system and facilitates communication between the software maintenance team members and the software maintenance manager.

There should be a centralized approval point for all software maintenance projects. This may be the software maintenance project manager or, for larger systems or organizations, a review board. The centralized approval process will enable one person or group of persons to have knowledge of all the requested and actual work being performed on the system. If this is not done, there is the likelihood that two or more independent changes to the system will be in conflict with one another and as a result, the system will not function properly. Additionally, different users will often request the same enhancements to a system but will have small differences in the details. By coordinating these requests, details can be combined and the total amount of resources required can be reduced.

Change request

There must be a formal, well-defined mechanism for initiating a request for changes or enhancements to a system. All changes considered for a system should be formally requested in writing. These requests may be initiated by the user or maintainer in response to discovered errors, new requirements, or changing needs. Procedures may vary regarding the format of a change request, but it is imperative that each request be fully documented in writing so it can be formally reviewed. Change requests should be carefully evaluated by the project manager or a change review board and decisions to proceed should be based on all the pertinent areas of consideration (probable effects on the system, actual need, resource requirements vs resource availability, budgetary considerations, priority, etc.). The decision and reasons for the decision should be recorded and included in the permanent documentation of the system.

Review and approval

Review and approval is the process of confirming that a software system meets the requirements and design specifications in the operational environment. It is a process which assures that the integral parts of the system perform according to the specifications.

Prior to installation, each change (correction, update, or enhancement) to a system should be formally reviewed. In practice, this process ranges from the review and sign-off by the project manager or user, to the convening of a change review board to formally approve or reject the changes. The purpose of this process is to ensure that all of the requirements of the change request have been met; that the system performs according to specifications; that the changes will not adversely impact the rest of the system or other users; that all procedures have been followed and rules and guidelines adhered to; and that the change is indeed ready for installation in the production system. All review actions and findings should be added to the system documentation folder.

Code Audit

Code review or audit is a procedure used to determine how well code adheres to established coding standards and practices and to the design specifications. The primary objective of code audits is to guarantee a high degree of uniformity across the software. This becomes a critical factor when someone other than the original developer must understand and maintain the software. Audits are also concerned with such program elements as commentary, labeling, paragraphing, initialization of common areas, and naming conventions. The audit should be performed by someone other than the original author.

4.2.5 Testing Standards and Procedures

Testing standards and procedures should define the degree and depth of testing to be performed and the disposition of test materials upon successful completion of the testing. Whenever possible, the test procedures and test data should be developed by someone other than the person who performed the actual maintenance on the system.

Testing is a critical component of software maintenance. As such the test procedures must be consistent and based on sound principles. The test plan should define the expected output and test for valid, invalid, expected, and unexpected cases. The test should examine whether or not the program is doing what it is supposed to do. The goal of testing is to find errors, not to prove that errors do not exist.

4.3 Managing Software Maintainers

Management is clearly one of the most important factors in improving the software maintenance process. Management must examine how the software is maintained, exercise control over the process, and ensure that effective software maintenance techniques and tools are employed. In order to maintain control over the software maintenance process and to ensure that the maintainability of a system does not deteriorate, it is important that software maintenance be anticipated and planned for.

The effective use of good management techniques and methodologies in dealing with scheduling maintenance, negotiating with users, coordinating the maintenance staff, and instituting the use of the proper tools and disciplines is essential to a successful software maintenance effort. Software maintenance managers are responsible for making decisions regarding the performance of software maintenance; assigning priorities to the requested work; estimating the level of effort for a task; tracking the progress of work; and assuring adherence to system standards in all phases of the maintenance effort. A software maintenance function has the same organizational needs and managerial problems as any other function.

Selecting the proper staff for a software maintenance project is as important as the techniques and approaches employed. While separate staffs for maintenance and development can improve the effectiveness of both, the realities of size, organization, budget, and staff ceilings often preclude the establishment of separate maintenance and development staffs.

Management must apply the same criteria to the maintainers that are applied to software and systems designers or other highly sought after professional positions. If an individual is productive, consistently performs well, has a good attitude, and displays initiative, it should not matter whether the project is development or maintenance. Three major psychological factors can impact the attitude, morale, and general performance of an individual:

- the work must be considered worthwhile by a set of values accepted by the individual, as well as by the standards employed by the organization.
- the individual must feel a responsibility for his or her performance. There is a need to feel personally accountable for the outcome of an effort.
- the individual must be able to determine on a regular basis whether or not the outcome of his or her efforts is satisfactory.

It is essential that work assignments offer growth potential. Continuing education is required at all levels to ensure that not only the maintainers, but the users, managers, and operators have a thorough understanding of software maintenance. Training should include: programming languages, standards and guidelines, operating systems, and utilities. Figure 3 outlines some points to keep in mind when staffing and managing a software maintenance function.

1. Maintenance is as important as development and just as difficult and challenging.
2. Maintainers should be highly qualified, competent, dedicated professionals. The staff should include both senior and junior personnel. Do not short change maintenance. Don't isolate the maintenance staff.
3. Maintenance should *NOT* be used as a training ground where junior staff are left to "sink-or-swim."
4. Staff members should be rotated so they are assigned to both maintenance and development.
5. Good maintenance performance and good development performance should be equally rewarded.
6. There should be an emphasis on keeping the staff well trained.
7. Rotate assignments. Do not permit a system or a major part of a system to become someone's private domain.

Figure 3. *Managing the software maintenance function*

5. SYSTEM MAINTENANCE VS SYSTEM REDESIGN

Although maintenance is an ongoing process, there comes a time when serious consideration should be given to redesigning a software system. A major concern of managers and software engineers is how to determine whether a system is hopelessly flawed or whether it can be successfully maintained. The costs and benefits of the continued maintenance of software which has become error-prone, ineffective, and costly must be weighed against that of redesigning the system.

When a decision has been reached to redesign or to stop supporting a system, the decision can be implemented in a number of ways. Support can simply be removed and the system can die through neglect; the minimum support needed to keep it functioning may be provided while a new system is built; or the system may be rejuvenated section by section and given an extended life. How the redesign is affected depends on the individual circumstances of the system, its operating environment, and the needs of the organization it supports.

While there are no absolute rules on when to rebuild rather than maintain the existing system, some of the factors to consider in weighing a decision to redesign or maintain are listed in figure 4. These characteristics are meant to be general "rules of thumb" which can assist a manager in understanding the problems in maintaining an existing system and in deciding whether or not it has outlived its usefulness to the organization. The greater the number of characteristics present, the greater the potential for redesign.

1. Frequent system failures
2. Code over 7 years old
3. Overly complex program structure and logic flow
4. Code written for previous generation hardware
5. Running in emulation mode
6. Very large modules or unit subroutines
7. Excessive resource requirements
8. Hard-coded parameters which are subject to change
9. Difficulty in keeping maintainers
10. Seriously deficient documentation
11. Missing or incomplete design specifications

Figure 4. *Characteristics of systems which are candidates for redesign*

Frequent System Failures

A system which is in virtually constant need of corrective maintenance is a prime candidate for redesign. As systems age and additional maintenance is performed, many become increasingly fragile and susceptible to changes. The older the code, the more likely frequent modifications, new requirements, and enhancements will cause the system to break down.

An analysis of errors should be made to determine whether the entire system is responsible for the failures, or if a few modules or sections of code are at fault. If the latter is found to be the case, then redesigning those parts of the system may suffice.

Code Over 7 Years Old

The estimated lifecycle of a major application system is 7-to-10 years. Software tends to deteriorate with age as a result of numerous fixes and patches. If a system is more than 7 years old, there is a high probability that it is outdated and expensive to run. A great deal of the code in use today falls into this category. After 7-to-10 years of maintenance, many systems have evolved to where additional enhancements or fixes are very time-consuming to make. A substantial portion of this code is probably neither structured, nor well-written. While this code was adequate and correct for the original environment, changes in technology and applications may have rendered it inefficient, difficult to revise, and in some cases obsolete. On the other hand, if the system was designed and developed in a systematic, maintainable manner, and if software maintenance was carefully performed and documented using established standards and guidelines, it may be possible to run it efficiently and effectively for many more years.

Overly Complex Program Structure and Logic Flow

"Keep it simple" must be the "golden rule" of all programming standards and guidelines. Too often, programmers engage in efforts to write a section of code in the least number of statements or utilizing the smallest amount of memory possible. This approach to coding usually results in complex code which is virtually incomprehensible. Poor program structure contributes to complexity. If the system being maintained contains a great deal of this type of code and the documentation is also severely deficient, it is a candidate for redesign.

Complexity also refers to the level of decision making present in the code. The greater the number of decision paths, the more complex the software is likely to be. Additionally, the greater the number of linearly independent control paths in a program, the greater the program complexity. Programs characterized by some or all of the following attributes are usually very difficult to maintain and are candidates for redesign:

- excessive use of DO loops
- excessive use of IF statements
- unnecessary GOTO statements
- embedded constants and literals
- unnecessary use of global variables
- self-modifying code
- multiple entry or exit modules
- excessive interaction between modules
- modules which perform same or similar functions.

Code Written for Previous Generation Hardware

Few industries have experienced as rapid a growth as the computer industry, particularly in the area of hardware. Not only have there been significant technological advances, but, the cost of hardware has decreased dramatically during the last decade. This phenomenon has generated a variety of powerful hardware systems. Software written for earlier generations of hardware is often inefficient on newer systems. Attempts to superficially modify the code to take advantage of the newer hardware is generally ineffective, time-consuming and expensive.

Running in Emulation Mode

One of the techniques used to keep a system running on newer hardware is to emulate the original hardware and operating system. Emulation refers to the capability of one system to exhibit behavior characteristic of another machine. In effect, it makes the host machine imitate the emulated machine. Emulation is normally used when resources are not available to convert a system, or the cost of doing so would be prohibitive. It frequently prevents utilization of the total capabilities and full power of the newer system. Emulated systems run a very fine line between functional usefulness and total obsolescence.

Very Large Modules or Unit Subroutines

"Mega-systems" which were written as one or several very large programs or sub-programs (thousands or tens-of-thousands of lines of code per program) can be extremely difficult to maintain. If the large modules can be restructured and divided into smaller, functionally related sections, the maintainability of the system will be improved.

Excessive Resource Requirements

An application system which requires a great deal of CPU time, memory, storage, or other system resources can place a very serious burden on all ADP users. Issues which must be addressed include whether it is cheaper to add more computer power or to redesign and reimplement the system, and whether redesign will reduce the resource requirements.

Hard-Coded Parameters Which Are Subject To Change

Many older systems were designed with the values of parameters used in performing specific calculations "hard coded" into the source code rather than stored in a table or read in from a data file. When changes in these values are necessary, (withholding rates, for example) each program in the system must be examined, modified and recompiled as necessary. This is a time-consuming, error prone process which is costly both in terms of the resources necessary to make the changes and the delay in getting the changes installed.

Whenever possible, the programs should be modified to handle the input of parameters in a single module or to read the parameters from a central table of values.

Difficulty in Keeping Maintainers

Programs written in low level languages, particularly assembler, require an excessive amount of time and effort to maintain. Generally, such languages are not widely taught or known. Therefore, it will be increasingly difficult to find maintainers who already know the language.

Seriously Deficient Documentation

Too often, documentation ranges from nonexistent to out-of-date. Even if the documentation is good when delivered, it often steadily and rapidly deteriorates as the software is modified. In some cases, the documentation is up-to-date, but still not useful. This can result when the documentation is produced by someone who does not understand the software or what is needed.

The worst documentation is that which is well-structured and formatted but which is incorrect or outdated. If there is no documentation, the maintainer will be forced to analyze the code in order to try to understand the system. If the documentation is physically deteriorated, the maintainer will be skeptical of it and verify its accuracy. If it looks good on the surface, but is technically incorrect, the maintainer may mistakenly believe it to be correct and accept what it contains. This will result in serious problems over and above those which originally necessitated the initial maintenance.

Missing or Incomplete Design Specifications

Knowing "how and why" a system works is essential to good maintenance. If the requirements and design specifications are missing or incomplete, the task of the maintainer will be more difficult. It is very important for the maintainer to not only understand what a system is doing, but how it is implemented, and why it was designed.

GLOSSARY

ADAPTIVE MAINTENANCE: Any effort which is initiated as a result of changes in the environment in which a software system must operate.

APPLICATION UTILITY LIBRARIES: See SOFTWARE LIBRARY.

APPLICATION SOFTWARE: Software specifically produced for the functional use of a computer system, for example payroll, general ledger, inventory control, human resources management.

BASELINE: A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development or maintenance, and that may be changed only through formal change control procedures.

BOTTOM-UP APPROACH: An approach that starts with the lowest level software components of a hierarchy and proceeds through progressively higher levels to the top level component.

CHANGE CONTROL: The process by which a change is proposed, evaluated, approved or rejected, scheduled, and tracked.

CHANGE REVIEW BOARD: The authority responsible for evaluating or disapproving proposed engineering changes, and ensuring implementation of the approved changes. Also referred to as the Configuration Control Board.

CHIEF PROGRAMMER: The leader of a chief programmer team. A senior-level programmer whose responsibilities include producing key portions of the software assigned to the team, coordinating activities of the team, reviewing work of the other team members, and having overall technical understanding of the software being developed or maintained.

CHIEF PROGRAMMER TEAM: A software development or maintenance group that employs support procedures designed to enhance group communication and to make optimum use of each member's skills.

CODE AUDIT: An independent review of source code by a person, team, or tool to verify compliance with software design, programming, and documentation standards.

CODE INSPECTION: The use of a formal set of procedures which are used to examine and measure the quality (error content) of the software.

COHESION: Cohesion refers to the degree to which the functions or processing elements within a module are related or bound together.

COMPILER: A computer program used to translate a high order language program into executable machine instructions.

COMPILER EXTENSION: Features of a programming language which are not included in the standard features (e.g., ANSI standard) of that language but are accepted and compiled by a specific compiler.

COMPLEXITY: The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics.

CORRECTIVE MAINTENANCE: Changes to a software system which are necessitated by actual errors (induced or residual) in a system.

COUPLING: The degree that modules are dependent upon each other in a computer program.

DESIGN REVIEW: The formal review of an existing or proposed design for the purpose of detection and remedy of design deficiencies and/or errors, and for the identification of possible improvements.

EMULATION: The imitation of all or part of one computer system by another so the imitating system accepts the same data, executes the same programs, and achieves the same results as the imitated system.

FIRMWARE: Computer programs and data loaded in a class of memory that cannot be dynamically modified by the computer during processing.

HIGH ORDER LANGUAGE (HOL): A programming language that does not reflect the logical structure of any one given computer or class of computers, for example, COBOL, FORTRAN, and PL/I.

LIBRARIAN: See SOFTWARE LIBRARIAN.

LIBRARY: See SOFTWARE LIBRARY.

LIFECYCLE: See SOFTWARE LIFECYCLE.

MAINTAINABILITY: The ease with which software can be maintained, for example, enhanced, adapted, or corrected to satisfy specified requirements.

MAINTENANCE: See SOFTWARE MAINTENANCE.

MODULAR: A program comprised of small, hierarchical units or sets of routines, where each performs a particular, unique function, is said to be modular.

MODULE: A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading, for instance, the input to, or output from, an assembler, compiler, linkage editor, or executive routine.

PEER REVIEW: A quality assurance method in which two or more peer programmers review and critique each other's work for accuracy and consistency with other parts of the system.

PERFECTIVE MAINTENANCE: All changes, insertions, deletions, modifications, extensions, and enhancements which are made to a system to meet the evolving and/or expanding needs of the user.

PROGRAM: A sequence of instructions suitable for processing by a computer.

REGRESSION TESTING: Rerunning test cases which a program has previously executed correctly to detect errors created during software maintenance.

SOFTWARE: Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.

SOFTWARE LIBRARIAN: The person responsible for establishing, controlling, and maintaining a software library.

SOFTWARE LIBRARY: A controlled collection of software and related documentation designed to aid in software development, use, or maintenance.

SOFTWARE LIFECYCLE: The period of time beginning when a software product is conceived and ending when the product is no longer available for use. The software lifecycle is typically broken into phases, such as requirements, design, implementation, testing, and operations and maintenance.

SOFTWARE MAINTENANCE: The performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production.

SOFTWARE TOOL: A computer program used to help develop, test, analyze, or maintain another computer program or its documentation.

SOURCE CODE: The program instructions written in a programming language.

STRUCTURED DESIGN: A disciplined approach to software design that adheres to a specified set of rules based on principles.

STRUCTURED PROGRAM: A program constructed of a basic set of control structures, each one having one entry point and one exit point. Less formally, any program which conforms to some disciplined approach intended to control the design, format, and logic structure of the program.

TESTING: Examining the behavior of a program by executing the program on sample data sets.

TOOL: See SOFTWARE TOOL.

TOP-DOWN APPROACH: An approach that starts with the highest level component of a hierarchy and proceeds through progressively lower levels.

VALIDATION: Determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements.

VERIFICATION: The demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development lifecycle.

VV&T: Validation, verification, and testing; used as an entity to define a procedure of review, analysis, and testing throughout the software lifecycle to discover errors, determine functionality, and ensure the production of quality software.

WALKTHROUGH: A manual analysis technique in which the module author describes the module's structure and logic to an audience of colleagues.

NOTE: Most of the definitions in this glossary appear in one or more of the following:

1. NBS Special Publication 500-106, "Guidance on Software Maintenance," by R. Martin and W. Osborne, December, 1983.
2. IEEE Computer Society, IEEE Std 729-1983, "IEEE Standard Glossary of Software Engineering Terminology," February, 1983.
3. Federal Information Processing Standard (FIPS) 11-2, "Guideline: American National Dictionary for Information Processing Systems," American National Standards Committee X3, Information Processing Systems, X3/TR-1-82, 1982.

APPENDIX I

Supporting ICST Documents

[FIPS38]	"Guidelines for Documentation of Computer Programs and Automated Data Systems," FIPS PUB 38, 1976.
[FIPS64]	"Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase," FIPS PUB 64, 1979.
[FIPS101]	"Guideline for Lifecycle Validation, Verification, and Testing of Computer Software," FIPS PUB 101, 1983.
[NBS56]	NBS Special Publication 500-56 "Validation, Verification, and Testing for the Individual Programmer," M. Branstad, J. Cherniavsky, and W. Adrion, 1980.
[NBS75]	NBS Special Publication 500-75 "Validation, Verification, and Testing of Computer Software," W. Adrion, M. Branstad, and J. Cherniavsky, 1981.
[NBS87]	NBS Special Publication 500-87 "Management Guide to Software Documentation," A. Neumann, 1982.
[NBS88]	NBS Special Publication 500-88 "Software Development Tools," R. Houghton, Jr., 1982.
[NBS93]	NBS Special Publication 500-93 "Software Validation, Verification, and Testing Technique and Tool Reference Guide," P. Powell, Editor, 1982.
[NBS106]	NBS Special Publication 500-106 "Guidance on Software Maintenance," R. Martin and W. Osborne, 1983.

Notes:

1. Subsequent NBS documents will include guidance on acceptance testing and other software engineering topics.

2. FIPS Guidelines documents may be ordered from:

   National Technical Information Service
   5285 Port Royal Road
   Springfield, VA 22161
   (703) 487-4650

3. NBS Special Publications may be ordered from:

   Superintendent of Documents
   U.S. Government Printing Office
   Washington, DC 20462
   (202) 783-3238

APPENDIX II

Suggested Additional Reading Material

[COUG82] D. J. Couger and M. A. Colter, "Effect of Task Assignments on Motivation of Programmers and Analysts," research report, University of Colorado, 1982.

[DONA80] J. Donahoo and D. Swearinger, "A Review of Software Maintenance Technology," Rome Air Development Center, RADC-TR-80-13, February 1980.

[GLAS79] R. L. Glass, *Software Reliability Guidebook,* Prentice-Hall, Englewood Cliffs, NJ, 1979.

[GLAS81] R. L. Glass and R. A. Noiseux, *Software Maintenance Guidebook,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[LIEN80] B. P. Lientz and E. B. Swanson, *Software Maintenance Management,* Addison-Wesley, Reading, MA, 1980.

[MART83] J. Martin, C. McClure, *Software Maintenance—The Problem and Its Solutions,* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[MCCL81] C. L. McClure, *Managing Software Development and Maintenance,* Van Nostrand Reinhold, NY, 1981.

[PARI80] G. Parikh, editor, *Techniques of Program and System Maintenance,* Ethnotech, Lincoln, NE, 1980.

[PARI83] G. Parikh, N. Zvegintzov, *Tutorial on Software Maintenance,* IEEE Computer Society Press, Silver Spring, MD, 1983.

[PERR81] W. E. Perry, *Managing System Maintenance,* Q.E.D. Information Sciences, Inc., Wellesley, MA, 1981.

[PRES82] R. Pressman, *Software Engineering: A Practitioner's Approach,* McGraw Hill, New York, 1982.

*U.S. GOVERNMENT PRINTING OFFICE 1984: 455-133/20060

NBS Technical Publications

Periodicals

Journal of Research—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year.

Nonperiodicals

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396).
NOTE: The Journal of Physical and Chemical Reference Data (JPCRD) is published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements are available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.
Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*
Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Service, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.